Quick Layered Response (QLR) Codes

Thomas Dean, Charles Dunn
Department of Electrical Engineering
Stanford University
Stanford, CA
Email: {trdean, ccdunn}@stanford.edu

***Abstract*—We propose a new 3D barcode standard, Quick Layered Response (QLR) codes, and implemented encoding and decoding on Android devices. QLR codes are Quick Response (QR) codes superimposed in the RGB color space, and increase the capacity by a factor of 3. This advantage can be used to decrease the area needed to represent data by a factor of 3 or to increase the readable distance. QLR codes can be encoded and decoded in linearly independent RGB basis colors, meaning attractive color schemes can be chosen. We also investigate the inclusion of modern error coding schemes in QLR codes.**

I. Introduction

Various forms of 2-dimensional (2D) barcodes have become popular due to their ease of readability and their storage capacity. Notable examples of 2D barcode standards include the QR code and the Data Matrix. It is also possible to make such barcodes colored, in effect increasing capacity through wavelength-division multiplexing. These 3-dimensional (3D) barcodes have been created in schemes such as SpectraCode, Mobile Multi-Colored Composite (MMCC) and Microsofts High Capacity Color Barcode (HCCB). Of these schemes, only the HCCB standard has gained traction, but is not aimed at increasing data capacity compared with black and white codes.

This project implements a method for color barcoding using Android devices as a reading platform, built on top of Google's ZXing ("Zebra Crossing") library. The new color barcode, called a Quick Layered Response (QLR, pronounced "color") Code, is three QR codes superimposed in different color channels. The corners of the code are used to correct color based on lighting conditions, and once color is corrected, the three layers are separated, sampled and binarized and decoded as standard QR codes. The color balancing technique also allows for any three colors to be used to construct the bar code, as long as the combination of their RGB values are linearly independent.

Fig. 1. [Left] QR code [Right] QLR code containing the same data

A. Related Work

A variety of attempts have been made at creating color barcoding schemes. MMCC codes were developed to target a variety of mobile cameras, including very low quality ones. HCCB codes are capable of using eight colors, but the basic version uses four which does not fully utilize the three independent color detectors of an Android camera. Other variants, like Paper Memory (PM) Codes, are overly sensitive to noise in the color spectrum. PM Codes claim to be able to use up to $17,000$ different colors to encode information, but this scheme would require overly complex error correction to account for the inevitably large reading bit error rate when using an Android to capture data. As a further difference, we are approaching the problem by viewing the color channels as three separate channels. In other words, we are not concerned with matching our eight (2^3 for three binary channels) color possibilities correctly, but rather we are concerned with properly deconstructing the individual channels to grayscale and then matching the values to the correct value. Our codes are based on the QR Code due to the fact that it is widely used. A similar process could be used to construct color bar codes

from most black and white barcodes.

II. ZXing Library & Development

From the ZXing website, "[ZXing] is an open-source, multi-format 1D/2D barcode image processing library implemented in Java, with ports to other languages. Our focus is on using the built-in camera on mobile phones to scan and decode barcodes on the device, without communicating with a server" [1]. ZXing is used as the foundation for numerous smart phone barcode reader applications, and also works on computers. It is capable of reading a dozen different barcode formats, including QR codes.

It should be noted that the ZXing Android app called Barcode Scanner only reads black and white bar codes; the first step in the image processing chain is to binarize the input image. We of course viewed this as throwing away $2/3$ of the available data since all camera phones use RGB sensors. This means that many of the classes used by the ZXing functions were insufficient to store full color data, but many of our added classes were simply arrays of the original black and white classes. Our final implementation includes all original ZXing functionality but with the additional capability of QLR code encoding, detection, and decoding. Our efforts also produced a useful zxtestbench java project that decodes QLR codes from input images.

III. QLR Codes on Android

Our QLR code encoding and decoding capability was succesfully implemented on top of the ZXing Barcode Scanner on Android devices.

A. Encoding

Generating QLR codes is done by layering three QR codes. The data to encode is padded to a multiple of three and then split into three codes of equal length. These are then individually encoded using ZXing's QR encoder. This means that any level of standard error correction (L,M,Q, or H) or even a combination for different color channels can also be used in QLR codes. Our QLR codes are designed to be shown on screens, so the standard color basis is pure red, green, and blue. The first third of the data is used for the red channel, the second for the green, and the last for the blue. Each channel has one of the corner alignment squares filled in. This distinguishes a QLR code from a QR code, gives the detector hints in order to balance the colors when reading the code, and identifies the order of the data when decoding. These modified QR codes are then combined in standard RGB format to create a single QLR code.

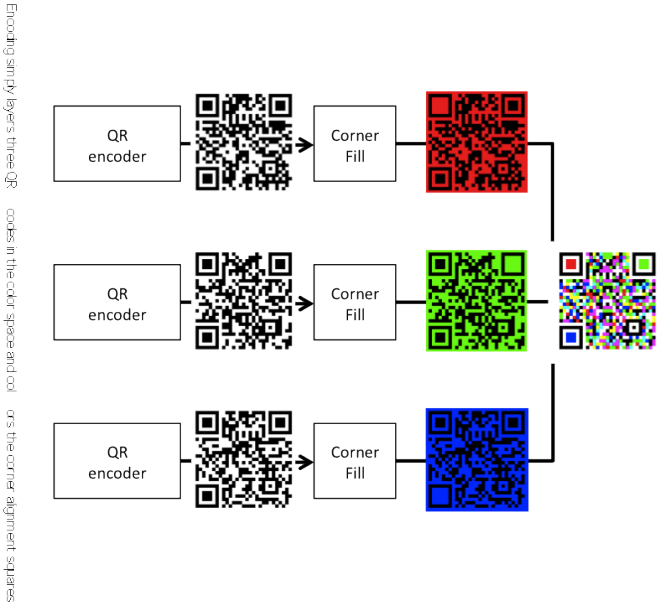

Fig. 2. Block diagram for encoding a QLR code

B. Detection

Detecting of QLR codes relies heavily on the QR detector library from ZXing. The QLR code is detected strictly from it's luminance values. The luminance values of each frame are first passed to a binarizor, which performs a local adaptive thresholding to return a binary bitmap of the frame. From here, the detector searches for corners. If three corners are found in a spacing that could be a QR code, the we assume that we have found a QR code. From here, we use the color values at the corner coordinates to correct and separate the color channels. The three channels are separated and binarized and passed to QR decoders.

1) Corner Detection: Corner detection algorithm is a key step to detecting QR or QLR codes. The algorithm was written by the authors of ZXing and not us. The top-left, top-right and bottom-right corners of any QR or QLR code is a black square is surrounding by a white and then black outline. In order to detect a corner, the detector simply counts across rows in a frame and looks for a ratio of black-white-black-white-black that is in proportion of 1-1-3-1-1. This algorithm is shift, scale and rotation invariant and requires very little computation. If three corners are found in an pattern that approximately forms a right triangle, we assume a code has been found

and we attempt to continue.

One disadvantage of using this technique is that the frame is thresholded based on the luminance values alone. Since green has a considerably higher luminance that blue and red, we encountered many cases where green was thresholded into white rather than black, causing the corner to be missed. This typically did not make the reader unable to read the code, but caused it to have many false attempts at decoding a valid frame before succeeding. There are many possible ways to fix this, including simply using a darker color (which will be corrected in the color balancing step) or thresholding based off of the minimum value of the RGB components.

2) Color Balancing: Upon detection of a QLR code based on the corners, the QR detector from ZXing returns the location of the corners from the luminance values. These corner locations are used to sample the RGB values and develop a basis in which the QLR code is represented in the camera image. In order to improve robustness, we included a color balancer that accounts for different lighting conditions, camera effects, and even codes printed using incorrect colors. Since all of the colors in the QLR code are combinations of the three corner colors and adding colors is linear, any affect on the corner alignment colors will affect any of the eight possible colors in the QLR code (assuming the affect is mostly constant across the QLR code). Due to this linearity, we can use a linear filter to do color balancing and convert any linearly independent combination of corner colors back to the standard pure red, green, and blue. In equation form, we have:

$$A \times [tl,\ tr,\ bl] = [r,\ g,\ b] \tag{1}$$

where A is our linear transformation and tl, tr, and bl are the top left, top right, and bottom left sampled corner colors, respectively.

$$A \times \begin{bmatrix} tl_R & tr_R & bl_R \\ tl_G & tr_G & bl_G \\ tl_B & tr_B & bl_B \end{bmatrix} = \begin{bmatrix} 255 & 0 & 0 \\ 0 & 255 & 0 \\ 0 & 0 & 255 \end{bmatrix}$$

$$A = \frac{1}{255} \begin{bmatrix} tl_R & tr_R & bl_R \\ tl_G & tr_G & bl_G \\ tl_B & tr_B & bl_B \end{bmatrix}^{-1}$$

Using a simple formula for the inverse of a matrix,

$$M^{-1} = \frac{adj(M)}{|M|} \tag{2}$$

where $adj(M)$ is the adjugate or adjoint matrix of M and $|M|$ is the determinant of M, we have a simple formula for calculating M^{-1} [2]. We therefore have a simple method of finding A. This was implemented in Java, as well as the matrix multiplication to convert all the pixels from the original image to a color balanced version.

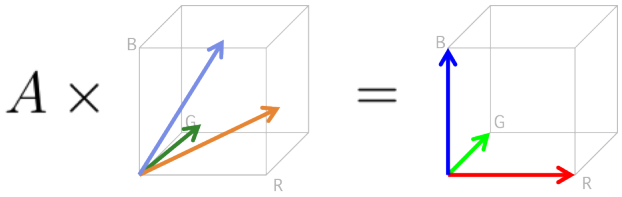

Fig. 3. Visual representation of orthogonalization in RGB domain

Taking a step back, what we have actually done is converted the components of the QLR code from some set of three basis vectors in RGB space to a set of three orthogonal basis vectors. As it turns out, we can not only correct for nonorthogonal colors due to distortion effects, but we can even correct for any combination of colors being completely switched in the image from the expected standard. Even more interestingly, we could actually print or display a QLR code in any three colors that are linearly independent in the RGB space, and still recover the data by converting this discolored QLR code to the correct colors. This attempt at improving robustness was certainly successful, and even allows for any number of color schemes for a decodable QLR code.

3) Separation and Sampling: After each color channel has been corrected, the next step is to separate the three channels. This is accomplished by simply creating an array of three objects that separately hold the red, green and blue values. Each of these objects can now be binarized in the same manner that the luminance values are.

We already know the locations of the corners of the QLR code. These corner values are used to create a transform that corrects for a shifted, rotated or skewed frame. Rather than applying this transform to the entire frame, it is used to properly sample each frame. Sampling is accomplished by dividing the frame into a grid and applying a maximal voting scheme on each square in the grid. This returns an approximation of the binary values stored with the code.

C. Decoding

The binarized, aligned data is now decodable in a fairly simple manner. The alignment squares in each individual channel are filled in to produce valid decodable QR codes, although this is not actually necessary since

these are used for detection, and have no effect on the data contained in a QR code. These three valid QR codes are each passed to a ZXing QR decoder, which produces the encoded data as a string after using Reed-Solomon decoding to rebuild any damaged bits. The three string outputs are simply concatenated in the correct order (red, green, then blue) and displayed on the android screen along with a binarized version of the capture image.

IV. Results

Our QLR codes performed extremely well in simulation, when feeding our detector and decoder with sample QLR codes. They also worked very well when detecting and decoding on Android devices. Our code works on the latest versions of Android devices, but especially well on the Transformer Prime tablet due to a better quality camera and display.

In designing the codes for phones, we wanted to codes to be as robust as possible, and in most senses we achieved this goal. We even created a ColorWheel Java Applet that rotated the color basis of a QLR code through combinations of colors. Because of our color balancing, the Android devices were still able to easily read the QLR code at any point in the Color Wheel. For example, even when the top left corner was cyan, the top right was purple, and the bottom left was yellow, we could still detect and decode the QLR code.

We did have occasional difficulties reading codes. One of the biggest issue was that green has a high luminance value, and therefor was often converted to white in the binarization process. To avoid this, we tried using darker colors of green with slightly more success. A better solution would be to pass the QR detector a minimum value of RGB instead of the luminance, so that no colored squares would be converted to white during binarization.

Another difficulty we realized was in reading very large codes. The Barcode Scanner app has an even smaller active frame region than the full camera, and so ensuring that the entire QLR code was inside the frame was difficult for larger codes, and made some impossible to decode. It should be noted that this problem is significantly worse with QR codes. In fact, QLR codes mediate the problem significantly by encoding data in a smaller area, so that the pixel size of the codes is still large when the camera is far enough away to fit the entire code. This is a major advantage of QLR codes over QR codes, especially when they are used for applications that would like to maximize readable distance; QLR codes can be decoded from much further away than QR codes where the code pixels simply become too small for a camera to sample properly.

V. Further Work

A. Analysis

More work should be done on quantitatively determining the error rates in our scheme. We would like to continue testing to determine from how far away QLR codes can be decoded, how non-orthogonal basis colors can be and still produce a decodable QLR code, and exactly what the error rate of decoding is. We can only currently state qualitatively that detecting a QLR code does not take prohibitively long, and that the decoding is quite fast once a code has been detected.

B. Flexible Coloring

Our color balancing led to some interesting possible color schemes. A great expansion would be encoding QLR codes in any combination of colors to make it more visually appealing, or even branding a QLR code by using a company's color scheme. Imagine a QLR for Google that was entirely composed of their classic red, yellow, green, and blue color scheme.

Taking this idea further, we could actually adapt the rate of our encoded data based on the orthogonality of the color scheme. Perfectly orthogonal basis colors would result in a high rate code, whereas nearly identical basis colors would result in a lower rate code. The outcome would be codes with different capacities, but similar error rates, regardless of the color scheme. This expansion of QLR codes with different encoding rates can be taken even further.

C. Advanced Error Correction Codes

In our proposal, we discussed the possibility of using a more modern iterative error-correction technique rather than the current Reed-Solomon scheme. We did not implement this for several reasons, one simply being time constraints. Adding color to the ZXing framework was a much more difficult task than we anticipated. Changing the error-correction routines that are being called would require a serious reworking of the entire framework, or possibly rebuilding of the detection process from scratch. It would also likely take a considerable amount of time required to get the required error-correcting routines working on the Android.

Another issue with using more modern coding techniques is the computational complexity of decoding. A Reed-Solomon code based on hard decisions, such as the one currently used in QR codes can easily be decoded

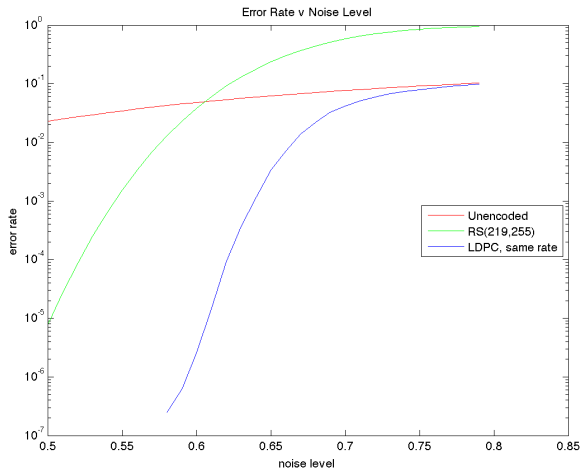

Fig. 4. Error rates for different encoding schemes

in real-time with very little computational power. By comparison, an LDPC code, which could be constructed to perform arbitrarily close to capacity, requires an incredible amount of computational power to be a useful form of error correction, and would likely not be a very usefully without specialized hardware.

The consequence of increased complexity in decoding would be a lag in the time required for the user to read a code. The current lag associated with reading the code is primarily caused by the decoder failing to properly decode frames. Using more robust error-correcting schemes would therefore cause a trade-off in performance.

The primary advantages of using a more robust scheme would be reduced overhead and a reduction in the number of unreadable frames. Figure 4 gives a comparison showing the potential increased performance in moving to an LDPC scheme with the same rate. The Reed-Solomon code used is the code used in the lowest level of error correction available in the QR code and has a rate of 0.86. Its curve is the performance of a theoretical hard-decision decoder over an AWGN channel [3]. The LDPC code of rate 0.85 over the same channel. This code was built by the authors and simulated using a decoding and framework built in C by the authors for EE388: Modern Coding Theory, and is based off [4]. It is difficult to tell to exactly what extent the rate of the LDPC code could be decreased to achieve a similar performance over the same channel. It should be noted that this comparison is over an AWGN channel, which may not adequately represent the channel in which the QLR codes are using. It should also be noted that the simulation of the LDPC code took many hours on a high-performance computer. Regardless, it shows that codes exist whose performance far exceeds that of a Reed-Solomon code.

Other possible error-correction schemes which have much lower computational complexity would likely be good candidates to be used in QLR codes, as they could provide significant gains in performance and still be possible to decode in a reasonable amount of time on the Android. Examples would include a turbo-product code based on the current Reed-Solomon codes [3]. Alternatively, using soft-bits with the current Reed-Solomon code and a list decoder would likely still result in a code that could be effectively decoded on the Android [5]. Furthermore, research should be performed to analyze the channel in which QLR codes are being read in order to properly tailor an error-correction scheme.

ACKNOWLEDGMENT

We would like to thank Professor Girod and the EE368 TAs, David Chen and Derek Pang, as well as our project advisor, Vijay Chandrasekhar. We would also like to acknowledge the creators of the ZXing library.

APPENDIX

A detailed workload distribution among the authors of this paper is given in Table I.

TABLE I
DETAILED WORKLOAD DISTRIBUTION

Task	Thomas	Charles
Planning	50 %	50 %
Code		
Encoding	5 %	95 %
ZXing Interface	90 %	20 %
Detection	85 %	15 %
Color Balancing	25 %	75 %
Decoding	95 %	5 %
Testing	80 %	20 %
Poster	15 %	85 %
Presentation	50 %	50 %
Report	20 %	80 %

REFERENCES

[1] ZXing ("Zebra Crossing"). 2012, http://code.google.com/p/zxing/
[2] "How to Inverse a 3X3 Matrix". 2012, http://www.wikihow.com/Inverse-a-3X3-Matrix

[3] S. Lin, D. Costello, Error Control Coding: Fundamentals and Applications. New York: Pearson-Prentice Hall, 2004.
[4] T. Richardson, R. Urbanke, Modern Coding Theory. New York: Cambridge University Press, 2009.
[5] S. Arora, B. Barak, Computational Complexity: A Modern Approach. New York: Cambridge, 2009, pp 393.